pFUnit 3.0 Tutorial
Basic

Tom Clune

Advanced Software Technology Group
Computational and Information Sciences and Technology Office
NASA Goddard Space Flight Center

April 10, 2014

Outline

Outline

1. Introduction
   - Overview
   - Quick review of testing
2. Introduction to pFUnit
3. References and Acknowledgements

Class Overview

Primary Goals:

- Learn how to use pFUnit 3.0 to create and run unit-tests
- Learn how to apply test-driven development methodology

Prerequisites:

- Access to Fortran compiler supported by pFUnit 3.0
- Familiarity with F95 syntax
- Familiarity with MPI[1]

Beneficial skills:

- Exposure to F2003 syntax - esp. OO features
- Exposure to OO programming in general

[1] MPI-specific sections can be skipped without impact to other topics.

Syllabus

- **Thursday PM - Introduction to pFUnit**
  - Overview of pFUnit and unit testing
  - Build and install pFUnit
  - Simple use cases and exercises
  - Detailed look at framework API
- **Friday AM - Advanced topics (including TDD)**
  - User-defined test subclasses
  - Parameterized tests
  - Introduction to TDD
  - Advanced exercises using TDD
- **Friday PM - Bring-your-own-code**
  - Incorporate pFUnit within the build process of your projects
  - Apply pFUnit/TDD in your own code
  - Supplementray exercises will be available

Materials

1. You will need access to one of the following Fortran compilers to do the hands-on portions
   - gfortran 4.9.0 (possibly available from cloud)
   - Intel 13.1, 14.0.2 (available on jellystone)
   - NAG 5.3.2
2. Last resort - use AWS
   - ssh keys are at `ftp://tartaja.com`
   - user name: `pfunit@tartaja.com` passwd: `iuse.PYTHON.1969`
   - login: `ssh-iuser1user1@54.209.194.237`
3. You will need a copy of the exercises in your work environment
   - Browser: `https://modelingguru.nasa.gov/docs/DOC-2529`
   - Jellystone: `/picnic/u/home/cacruz/pFUnit.tutorial/Exercises.tar`
4. These slides can be downloaded at `https://modelingguru.nasa.gov/docs/DOC-2528`

Outline

Quick review of testing

- What is a (software) test?
- What is a unit test?
- What are desirable properties for unit tests?
- What is the anatomy of a unit test?
- What is a test "fixture"

A test by any other name ...

A test is *any* mechanism that can be used to verify a software implementation. Examples include:

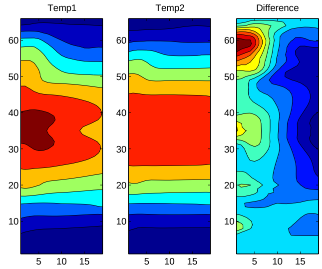
- Conditional termination during execution:

```
IF (PA(I,J)+PTOP.GT.1200.) &
   call stop_model('ADVECM: Pressure diagnostic
```

- Diagnostic print statement

```
print*, 'loss of mass = ', deltaMass
```

- Inspection of rendered output:



What is a unit test?

What is a unit test?

> *"A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work."*
>
> — The Art of Unit Testing

What is a unit test?

> *"A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work."*
>
> — The Art of Unit Testing

For our purposes a *unit* is a single Fortran subroutine or function.

Desirable attributes for tests:

Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.

Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.

Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
- Complete
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*

Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
- Complete
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
- Independent - *No side effects*
  - No STDOUT; temp files deleted; ...
  - Order of tests has no consequence.
  - Failing test does *not* terminate execution.

Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
- Complete
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
- Independent - *No side effects*
  - No STDOUT; temp files deleted; ...
  - Order of tests has no consequence.
  - Failing test does *not* terminate execution.
- Frugal
  - Execute quickly (think 1 millisecond)
  - Small memory, etc.

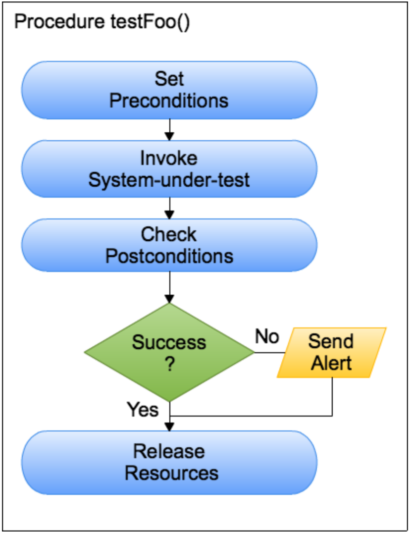
Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
- Complete
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
- Independent - *No side effects*
  - No STDOUT; temp files deleted; ...
  - Order of tests has no consequence.
  - Failing test does *not* terminate execution.
- Frugal
  - Execute quickly (think 1 millisecond)
  - Small memory, etc.
- Automated and repeatable

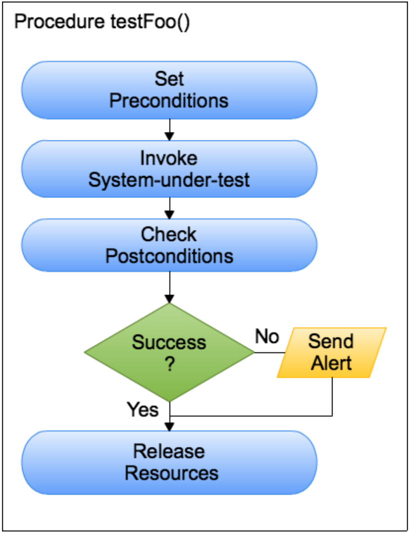
Desirable attributes for tests:

- Narrow/specific
  - Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
  - Each defect causes failure in one or only a few tests.
- Complete
  - All functionality is covered by at least one test.
  - *Any defect is detectable.*
- Independent - *No side effects*
  - No STDOUT; temp files deleted; ...
  - Order of tests has no consequence.
  - Failing test does *not* terminate execution.
- Frugal
  - Execute quickly (think 1 millisecond)
  - Small memory, etc.
- Automated and repeatable
- Clear intent

Anatomy of a Software Test Procedure

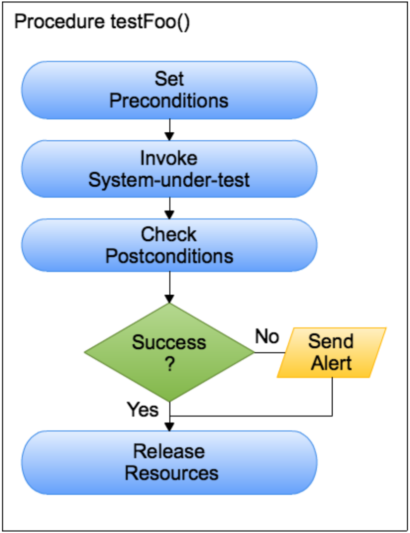


Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

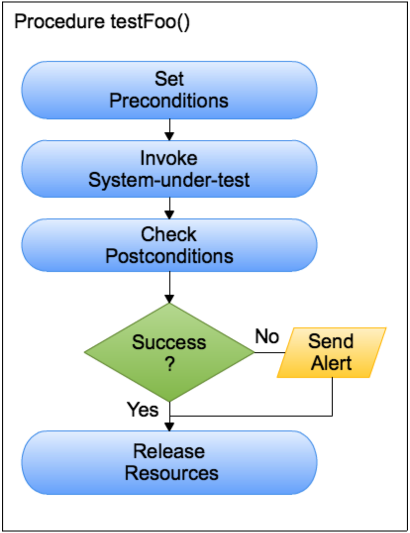
Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

a = 2.; t = 3.

Anatomy of a Software Test Procedure

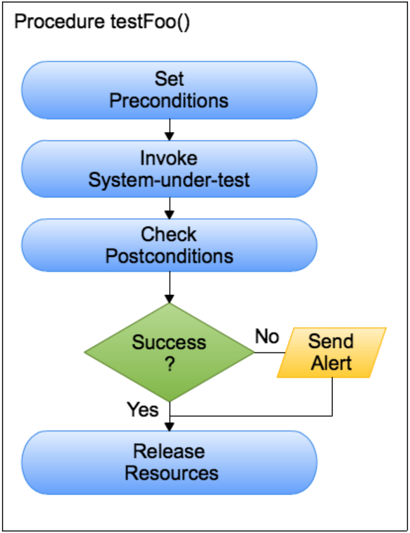


testTrajectory() ! $s = \frac{1}{2}at^2$

a = 2.; t = 3.

s = trajectory(a, t)

Anatomy of a Software Test Procedure



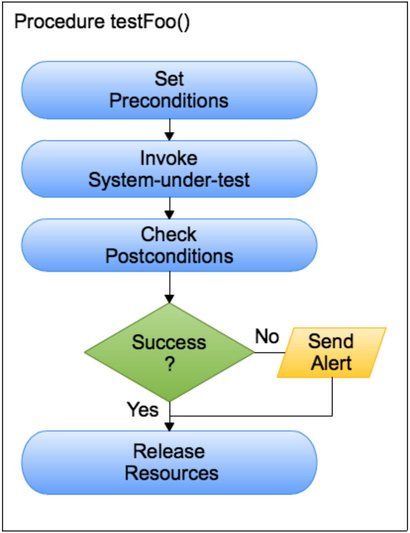
testTrajectory() ! $s = \frac{1}{2}at^2$

a = 2.; t = 3.

s = trajectory(a, t)

```
call assertEqual(9., s)
```

Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

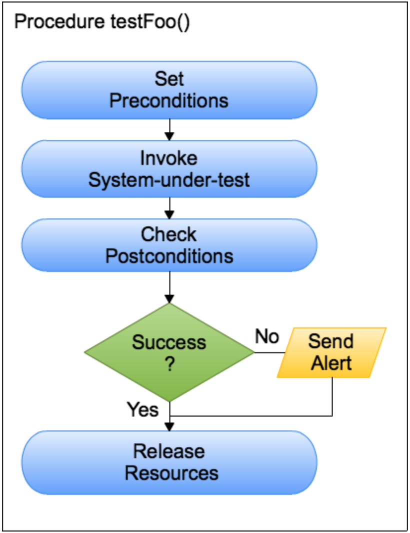
a = 2.; t = 3.

s = trajectory(a, t)

```
call assertEqual(9., s)
```

! no op

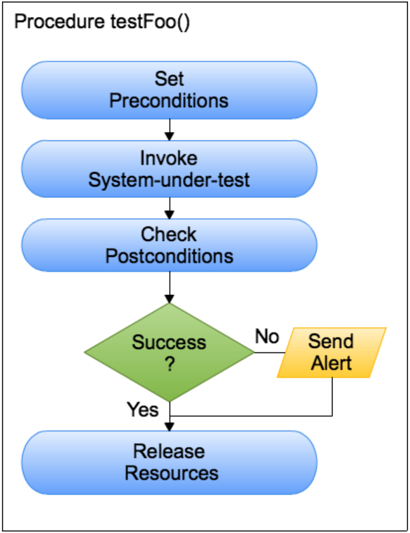
Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

```
call assertEqual(9., trajectory(2.,3.))
```

Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

```
@assertEqual(9., trajectory(2.,3.))
(automatically includes
file name and line number)
```

Outline

Outline

Brief history of pFUnit

2005 First prototype[1]

- Re-implemented using TDD after reading a book

2006 Version 1.0 released as open source

2012 Began serious attempt at F2003/OO implementation[2]

2013 Version 2.0 released - heavy reliance of OO

- New and improved parser (test "annotations")
- Numerous new assertions

2014 Version 3.0 released[3]

- Introduced build with cmake
- Custom test cases finally "easy"
- Driver command line options (–debug, -o, –xml)

[1]Proof to colleague that Fortran (F90) could do this (I cheated)

[2]Great joy navigating immature compilers.

[3]Would have been 2.2, but bug in gfortran broke backwards compatibility

Noteworthy features of pFUnit 3.0

[1] F2003 with a dash of F2008

[2] Threadsafe

Noteworthy features of pFUnit 3.0

- Implemented in standard Fortran[1]

[1] F2003 with a dash of F2008
[2] Threadsafe

Noteworthy features of pFUnit 3.0

- Implemented in standard Fortran[1]
- Has strong support for multidimensional arrays

[1] F2003 with a dash of F2008

[2] Threadsafe

Noteworthy features of pFUnit 3.0

- Implemented in standard Fortran[1]
- Has strong support for multidimensional arrays
- Enables testing of parallel applications - MPI & OpenMP[2]

[1] F2003 with a dash of F2008

[2] Threadsafe

Noteworthy features of pFUnit 3.0

- Implemented in standard Fortran[1]
- Has strong support for multidimensional arrays
- Enables testing of parallel applications - MPI & OpenMP[2]
- Enables custom test fixtures

[1] F2003 with a dash of F2008

[2] Threadsafe

Noteworthy features of pFUnit 3.0

- Implemented in standard Fortran[1]
- Has strong support for multidimensional arrays
- Enables testing of parallel applications - MPI & OpenMP[2]
- Enables custom test fixtures
- Enables parameterized tests

[1] F2003 with a dash of F2008

[2] Threadsafe

Noteworthy features of pFUnit 3.0

- Implemented in standard Fortran[1]
- Has strong support for multidimensional arrays
- Enables testing of parallel applications - MPI & OpenMP[2]
- Enables custom test fixtures
- Enables parameterized tests
- Extensible via OO features of Fortran

[1] F2003 with a dash of F2008

[2] Threadsafe

Noteworthy features of pFUnit 3.0

- Implemented in standard Fortran[1]
- Has strong support for multidimensional arrays
- Enables testing of parallel applications - MPI & OpenMP[2]
- Enables custom test fixtures
- Enables parameterized tests
- Extensible via OO features of Fortran
- Greatly improves usability via elegant preprocessor annotations

[1] F2003 with a dash of F2008

[2] Threadsafe

Noteworthy features of pFUnit 3.0

- Implemented in standard Fortran[1]
- Has strong support for multidimensional arrays
- Enables testing of parallel applications - MPI & OpenMP[2]
- Enables custom test fixtures
- Enables parameterized tests
- Extensible via OO features of Fortran
- Greatly improves usability via elegant preprocessor annotations
- Contains improved (and maintained!) examples

[1] F2003 with a dash of F2008

[2] Threadsafe

Noteworthy features of pFUnit 3.0

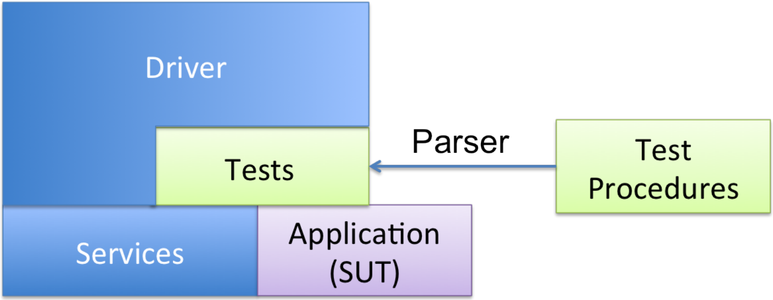
- Implemented in standard Fortran[1]
- Has strong support for multidimensional arrays
- Enables testing of parallel applications - MPI & OpenMP[2]
- Enables custom test fixtures
- Enables parameterized tests
- Extensible via OO features of Fortran
- Greatly improves usability via elegant preprocessor annotations
- Contains improved (and maintained!) examples
- Covered by regression self-tests after each push

[1] F2003 with a dash of F2008
[2] Threadsafe

Useful resources

- Website/documentation http://pfunit.sourceforge.net
  - somewhat out of date
- Mailing list: pfunit-support@lists.sourceforge.net
- This tutorial https://modelingguru.nasa.gov/docs/DOC-2528
- Contact me: Thomas.L.Clune@nasa.gov

pFUnit Architecture



Outline

Supported compilers[1,2]

OS	Vendor		Version
Linux	Intel	ifort	14.0.2
Linux	Intel	ifort	13.1.192
Linux	GNU	gfortran	4.9.0[3]
Linux	NAG	nagfor	5.3.2(981)
OS X	Intel	ifort	14.0.2
OS X	GNU	gfortran	4.9.0[3]
OS X	NAG	nagfor	5.3.2(979)
AIX	IBM	xlf	???
Windows	Intel	ifort	???

Covered by regression tests

External contributions

[1] In many cases closely related compiler versions will also work.
[2] We are cautiously optimistic that PGI will soon be supported.
[3] Not yet released. 4.9.0 trunk does work, and 4.8.3 is expected to work.

Misc requirements

python	2.6+
MPI	2.0+[1]
git	1.7.1[2]
gmake	3.8.1
CMake	2.8.10

[1] Probably even 1.3

[2] Earlier verrsions may have issues with branching

Obtaining pFUnit

For this discussion we will refer to 3 distinct directories:

- root - top directory of downloaded code
- build - directory in which build instructions are issued
- install - directory where various framework elements will be installed for later use.

There are 2 ways to obtain the source code for pFUnit:

- Via git (read-only):
  `git clone git://git.code.sf.net/p/pfunit/code pFUnit`
- Via tar:
  `http://sourceforge.net/projects/pfunit/files/latest/download`
  `tar zxf ./pFUnit.tar.gz`

Exercise 0: Installation (download)

Step 0: Change directory to parent of `<root_dir>`

Step 1: Download source

- **git**: `% git clone git://git.code.sf.net/p/pfunit/code pFUnit`
- **tar**:
  - `http://sourceforge.net/projects/pfunit/files/latest/download`
  - `mv <download_dir>/pFUnit.tar.gz <root_dir>/..`
  - `cd <root_dir>/..`
  - `tar -xzf ./pFUnit.tar.gz`

Exercise 0: Installation (download)

Step 0: Change directory to parent of `<root_dir>`

Step 1: Download source

- **git**: `% git clone git://git.code.sf.net/p/pfunit/code pFUnit`
- **tar**:
  - `http://sourceforge.net/projects/pfunit/files/latest/download`
  - `mv <download_dir>/pFUnit.tar.gz <root_dir>/..`
  - `cd <root_dir>/..`
  - `tar -xzf ./pFUnit.tar.gz`

Synchronize attendees ...

What's in the distribution?

```
bash-3.2$ cd pFUnit
bash-3.2$ dir
total 104
16 CMakeLists.txt        24 LICENSE              0 source/
 8 COPYRIGHT             32 README-INSTALL       0 tests/
 8 Copyright.txt          0 bin/                 0 tools/
 0 Examples/              0 documentation/
16 GNUmakefile            0 include/
```

Exercise 0: Compile, test, install (serial)

Step 2: Compile serial

1. `setenv PFUNIT <serial_install_dir>`
2. Choose build option
   - **gmake**
     1. `cd <root_dir>`
     2. `make -j tests F90_VENDOR=Intel F90=ifort`
     3. `make install INSTALL_DIR=$PFUNIT`
   - **cmake**
     1. `cd <root_dir>`
     2. `mkdir build_serial`
     3. `cd build_serial`
     4. `cmake ..`
     5. `make -j tests`
     6. `make install`

Exercise 0: Compile, test, install (serial)

Step 2: Compile serial

1. `setenv PFUNIT <serial_install_dir>`
2. Choose build option
   - **gmake**
     1. `cd <root_dir>`
     2. `make -j tests F90_VENDOR=Intel F90=ifort`
     3. `make install INSTALL_DIR=$PFUNIT`
   - **cmake**
     1. `cd <root_dir>`
     2. `mkdir build_serial`
     3. `cd build_serial`
     4. `cmake ..`
     5. `make -j tests`
     6. `make install`

Synchronize attendees ...

Exercise 0: Compile, test, install (MPI)

Step 3: Compile MPI

1. `setenv PFUNIT <mpi_install_dir>`
2. Choose build option
   - **gmake**
     1. `cd <root_dir>`
     2. `make -j tests MPI=YES F90_VENDOR=Intel F90=ifort`
     3. `make install INSTALL_DIR=$PFUNIT`
   - **cmake**
     1. `cd <root_dir>`
     2. `mkdir build_mpi`
     3. `cd build_mpi`
     4. `cmake .. -DMPI=YES`
     5. `make -j tests`
     6. `make install`

Exercise 0: Compile, test, install (MPI)

Step 3: Compile MPI

1. `setenv PFUNIT <mpi_install_dir>`
2. Choose build option
   - **gmake**
     1. `cd <root_dir>`
     2. `make -j tests MPI=YES F90_VENDOR=Intel F90=ifort`
     3. `make install INSTALL_DIR=$PFUNIT`
   - **cmake**
     1. `cd <root_dir>`
     2. `mkdir build_mpi`
     3. `cd build_mpi`
     4. `cmake .. -DMPI=YES`
     5. `make -j tests`
     6. `make install`

Synchronize attendees ...

Cheat for jellystone users

Install pFUnit using the pFUnit installer executing the following script:
`/picnic/u/home/cacruz/pFUnit.tutorial/install.pFUnit`
The script will install pFUnit in a location of your choice.

Alternatively, users can access pre-installed installations in
`/picnic/u/home/cacruz/pFUnit.tutorial`
To use these set the PFUNIT environment variable as follows: csh (bash)

- Base directory

```
setenv PFUNIT_BASE /picnic/u/home/cacruz/pFUnit.tutorial
(export PFUNIT_BASE=/picnic/u/home/cacruz/pFUnit.tutorial)
```

- For serial exercises:

```
setenv PFUNIT $PFUNIT_BASE/pFUnit.serial
(export PFUNIT=$PFUNIT_BASE/pFUnit.serial)
```

- For MPI exercises:

```
setenv PFUNIT $PFUNIT_BASE/pFUnit.mpi
(export PFUNIT=$PFUNIT_BASE/pFUnit.mpi)
```

Outline

Simple Example: testing factorial function

Here is a simple unit test that checks $2! = 2$

```
@test
subroutine testFactorial2()
   use pFUnit_mod
   use Factorial_mod
   @assertEqual(2, factorial(2))
end subroutine testFactorial2
```

./Exercises/SimpleTest/testFactorialA.pf

Simple Example: testing factorial function

Here is a simple unit test that checks $2! = 2$

```
@test
subroutine testFactorial2()
   use pFUnit_mod
   use Factorial_mod
   @assertEqual(2, factorial(2))
end subroutine testFactorial2
```

./Exercises/SimpleTest/testFactorialA.pf

- Note: This is *not* standard conforming Fortran.
- File must be preprocessed prior to compilation.

Simple Example: testing factorial function

Here is a simple unit test that checks $2! = 2$

```
@test
subroutine testFactorial2()
   use pFUnit_mod
   use Factorial_mod
   @assertEqual(2, factorial(2))
end subroutine testFactorial2
```

./Exercises/SimpleTest/testFactorialA.pf

Procedure must "USE" `pFUnit_mod`.

- Imports all pFUnit derived-types and procedures
- E.g., the assertion routines

Simple Example: testing factorial function

Here is a simple unit test that checks $2! = 2$

```
@test
subroutine testFactorial2()
   use pFUnit_mod
   use Factorial_mod
   @assertEqual(2, factorial(2))
end subroutine testFactorial2
```

./Exercises/SimpleTest/testFactorialA.pf

Test procedures are indicated by the `@test` annotation.

- Must immediately precede `subroutine` declaration
- Preprocessor generates code to "register" the test procedure

Simple Example: testing factorial function

Here is a simple unit test that checks $2! = 2$

```
@test
subroutine testFactorial2()
   use pFUnit_mod
   use Factorial_mod
   @assertEqual(2, factorial(2))
end subroutine testFactorial2
```

./Exercises/SimpleTest/testFactorialA.pf

Expected results are indicated with the `@assertEqual` annotation.

- First argument is *expected* value
- Second argument is *found* value

Simple Example: testing factorial function

Here is a simple unit test that checks $2! = 2$

```
@test
subroutine testFactorial2()
   use pFUnit_mod
   use Factorial_mod
   @assertEqual(2, factorial(2))
end subroutine testFactorial2
```

./Exercises/SimpleTest/testFactorialA.pf

`@assertEqual` expands to:

```
  call assertEqual(2, factorial(2), &
 & location=SourceLocation( &
 & 'testFactorialA.pf', &
 & 5) )
  if (anyExceptions()) return
#line 6 "testFactorialA.pf"
```

Simple Example: testing factorial function

Suites of tests must be registered with the `pFUnit` driver through a special file called 'testSuites.inc':

For the factorial example we have[3]

```
! Register your test suites here
ADD_TEST_SUITE(testFactorialA_suite)
!ADD_TEST_SUITE(testFactorialB_suite)
```

Simple Example: testing factorial function

Suites of tests must be registered with the pFUnit driver through a special file called 'testSuites.inc':

For the factorial example we have[3]

```
! Register your test suites here
ADD_TEST_SUITE(testFactorialA_suite)
!ADD_TEST_SUITE(testFactorialB_suite)
```

The parser generates one test suite per file/module.
Suite names are derived from the file containing the tests.

- For modules, default suite name is <module_name>_suite
- Otherise default suite name is <file_name>_suite
- Can override with @suite=<name> annotation

Simple Example: testing factorial function

Suites of tests must be registered with the pFUnit driver through a special file called 'testSuites.inc':

For the factorial example we have[3]

```
! Register your test suites here
ADD_TEST_SUITE(testFactorialA_suite)
!ADD_TEST_SUITE(testFactorialB_suite)
```

'testSuites.inc' has the following structure

- One entry per test suite
- Each entry is of the form of a CPP macro `ADD_TEST_SUITE(<suite_name>)`
- Macro is **case sensitive**

Simple Example: testing factorial function

Suites of tests must be registered with the `pFUnit` driver through a special file called 'testSuites.inc':

For the factorial example we have[3]

```
! Register your test suites here
ADD_TEST_SUITE(testFactorialA_suite)
!ADD_TEST_SUITE(testFactorialB_suite)
```

In this example there is one *active* test suite:

- `testFactorialA_suite` from file 'testFactorialA.F90'

Simple Example: testing factorial function

Suites of tests must be registered with the `pFUnit` driver through a special file called 'testSuites.inc':

For the factorial example we have[3]

```
! Register your test suites here
ADD_TEST_SUITE(testFactorialA_suite)
!ADD_TEST_SUITE(testFactorialB_suite)
```

`testFactorialB_suite` from file 'testFactorialB.F90' is *inactive*:

- Fortran comment character ('!') at the beginning of the line prevents registration.

Simple Example: testing factorial function

Here is the simple makefile[4] for our example:

```
.PHONY: tests clean
%.F90: %.pf
	$(PFUNIT)/bin/pFUnitParser.py $< $@ -I.
TESTS = $(wildcard *.pf)

%.o : %.F90
	$(FC) -c $< -I$(PFUNIT)/mod
SRCS = $(wildcard *.F90)

OBJS = $(SRCS:.F90=.o) $(TESTS:.pf=.o)
DRIVER = $(PFUNIT)/include/driver.F90

tests.x: $(DRIVER) $(OBJS) testSuites.inc
	$(FC) -o $@  -I$(PFUNIT)/mod $^ -L$(PFUNIT)/lib -lpfunit -I.
tests: tests.x
	./tests.x

clean:
	$(RM) *.o *.mod *.x *~
```

The first rule above shows the invocation of the python script called ‘pFUnitParser.py’.

[4] From ./Exercises/SimpleTest/GNUmakefile

Simple Example: testing factorial function (contd́)

Checklist for simple tests:

1. Each test is preceded by `@test`
2. Each test *file* has corresponding line in `testSuites.inc`:

   ```
   ADD_TEST_SUITE(<suite>)
   ```

3. Makefile must know to preprocess test files:

   ```
   %.pf : %.F90
           $(PFUNIT)/bin/pFUnitParser.py $< $@
   ```

A bit more on annotations

The pFUnit parser has a very inflexible syntax:[5]

- Each annotation must be on a single line[6]
- No end-of-line comment characters
- Comment at beginning of line deactivates that annotation

[5] I am not in the business of automatically parsing Fortran.

[6] I expect this to be relaxed in the future.

A bit more on annotations

The pFUnit parser has a very inflexible syntax:[5]

- Each annotation must be on a single line[6]
- No end-of-line comment characters
- Comment at beginning of line deactivates that annotation

Also are some restrictions on style for intermingled Fortran:

- Only supports free-format. (Fixed-format application code is fine.)
- Test procedure declarations must be on one line:

```
@test
subroutine testA()
```

Correct

[5] I am not in the business of automatically parsing Fortran.

[6] I expect this to be relaxed in the future.

A bit more on annotations

The pFUnit parser has a very inflexible syntax:[5]

- Each annotation must be on a single line[6]
- No end-of-line comment characters
- Comment at beginning of line deactivates that annotation

Also are some restrictions on style for intermingled Fortran:

- Only supports free-format. (Fixed-format application code is fine.)
- Test procedure declarations must be on one line:

```
@test
subroutine testA()
```

Correct

```
@test
subroutine &
 & testA()
```

Illegal - must be on one line

[5] I am not in the business of automatically parsing Fortran.

[6] I expect this to be relaxed in the future.

pFUnit Driver

The driver is a short program that

- bundles the various test suites into a single suite
- runs the tests
- produces a short summary

Users can write their own if desired.[7]

Driver uses F2003 feauters to provide the following command line support

- `-h, --help` display options
- `-v, --verbose -d --debug` more reporting
- `-o <file>` rerout output
- `-robust` not reliable at this time
- `-skip` used internally (with `-robust`)

[7] Requires some knowledge of F2003, and advanced aspects of pFUnit.

Exercise 1a: Build the "SimpleTest" example in the distribution

1. Change directory to `./Exercises/SimpleTest`
2. set `$PFUNIT` to the *serial* installation
3. `make tests`
4. Verify that 1 test ran successfully.

If successful you should see something like:

```
.
Time:         0.002 seconds

 OK
 (1 test)
```

Exercise 1b: Activate other test file

1. Edit `./testSuites.inc` and uncomment the 2nd suite
2. `make tests`

You should see something like:

```
..
Time:         0.006 seconds

OK
(2 tests)
```

Look at the test file.
Question: Why are there 2 tests rather than 3?
Notice the periods: there is one for each test run.[1]

[1] Useful for large collections to show that the tests are proceeding.

Exercise 1c: Activate 3rd test

1. Edit the file `./Exercises/SimpleTest/testFactorialB.pf`: and insert the `@test` annotation before the 2nd test procedure
2. `make tests`

You should see something like:

```
...
Time:         0.006 seconds

OK
(3 tests)
```

Exercise 1d: Create a 4th test

1. Create a new test procedure that verifies $0! = 1$
2. `make tests` (uh oh!)

```
....F
Time:          0.002 seconds

 Failure in: testFactorialD
   Location: [testFactorial.pf:26]
expected: <1> but found: <0>

 FAILURES!!!
Tests run: 4, Failures: 1, Errors: 0
*** Encountered 1 or more failures/errors during testing. **
make: *** [tests] Error 128
```

3. Fix the implementation
4. `make tests` (whew!)

Exercise 1e: Demonstrate tests as a harness

1. Edit `factorial.F90`
2. Insert a bug (e.g., change '*' to '+')
3. `% make tests`

Testing MPI-based procedures

Introduces new ways to fail

Testing MPI-based procedures

Introduces new ways to fail

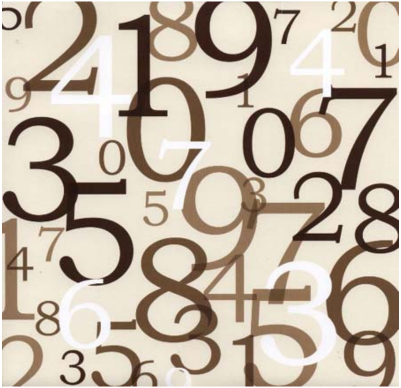
- **Fail on any/all processes**

www.glogster.com

Testing MPI-based procedures

Introduces new ways to fail

- Fail on any/all processes
- **Fail when varying number of processes**

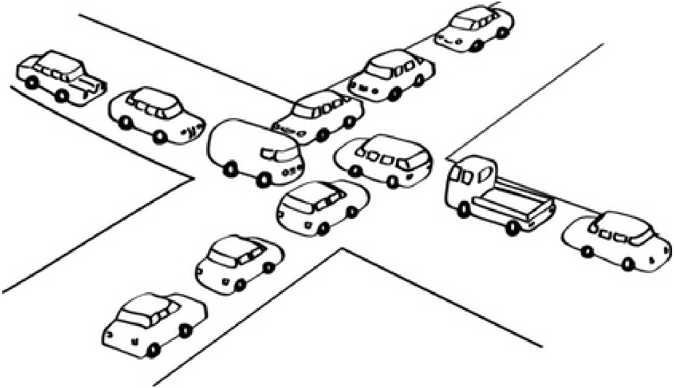


www.thebigquestions.com

Testing MPI-based procedures

Introduces new ways to fail

- Fail on any/all processes
- Fail when varying number of processes
- **Race, deadlock, ...**



csunplugged.org

Basic MPI example: matrix transpose

```
@test(npes=[1])
! Transpose of 1x1 is just identity.
subroutine testTranspose_trivial(this)
   class (MpiTestMethod), intent(inout) :: this
   real :: a(1,1), at(1,1)
   integer :: comm

   a = 1 ! preconditions

   comm = this%getMpiCommunicator()
   call transpose(comm, a, at)

   @mpiAssertEqual(1, at)
end subroutine testTranspose_trivial
```

./Exercises/SimpleMpiTest/TestTranspose.pf

Basic MPI example: matrix transpose

```fortran
  @test(npes=[1])
  ! Transpose of 1x1 is just identity.
  subroutine testTranspose_trivial(this)
     class (MpiTestMethod), intent(inout) :: this
     real :: a(1,1), at(1,1)
     integer :: comm

     a = 1 ! preconditions

     comm = this%getMpiCommunicator()
     call transpose(comm, a, at)

     @mpiAssertEqual(1, at)
  end subroutine testTranspose_trivial
```

./Exercises/SimpleMpiTest/TestTranspose.pf

@test accepts an optional argument npes=[<list>]

- Indicates to framework that test procedure uses MPI.
- Test procedure will execute once for each item in < list >
- New subcommunicator of indicated size created for each execution

Basic MPI example: matrix transpose

```
@test(npes=[1])
! Transpose of 1x1 is just identity.
subroutine testTranspose_trivial(this)
   class (MpiTestMethod), intent(inout) :: this
   real :: a(1,1), at(1,1)
   integer :: comm

   a = 1 ! preconditions

   comm = this%getMpiCommunicator()
   call transpose(comm, a, at)

   @mpiAssertEqual(1, at)
end subroutine testTranspose_trivial
```

./Exercises/SimpleMpiTest/TestTranspose.pf

MPI tests have a single, mandatory argument

- Used by framework to pass MPI context information
- TYPE and INTENT must be exactly as above
- Keyword CLASS is an F2003 OO extension

Basic MPI example: matrix transpose

```
  @test(npes=[1])
  ! Transpose of 1x1 is just identity.
  subroutine testTranspose_trivial(this)
     class (MpiTestMethod), intent(inout) :: this
     real :: a(1,1), at(1,1)
     integer :: comm

     a = 1 ! preconditions

     comm = this%getMpiCommunicator()
     call transpose(comm, a, at)

     @mpiAssertEqual(1, at)
  end subroutine testTranspose_trivial
```

./Exercises/SimpleMpiTest/TestTranspose.pf

Mandatory argument is an *object* with useful methods

```
comm = this%getMpiCommunicator()
npes = this%getNumProcesses()
rank = this%getProcessRank()
```

Basic MPI example: matrix transpose

```
@test(npes=[1])
! Transpose of 1x1 is just identity.
subroutine testTranspose_trivial(this)
   class (MpiTestMethod), intent(inout) :: this
   real :: a(1,1), at(1,1)
   integer :: comm

   a = 1 ! preconditions

   comm = this%getMpiCommunicator()
   call transpose(comm, a, at)

   @mpiAssertEqual(1, at)
end subroutine testTranspose_trivial
```

./Exercises/SimpleMpiTest/TestTranspose.pf

@mpiAssertEqual is a variant of **@assertEqual**

- Enforces synchronization among processes
- *Both* forms will attach information about rank and npes in MPI test

Basic MPI example: matrix transpose

```
@test(npes=[1])
! Transpose of 1x1 is just identity.
subroutine testTranspose_trivial(this)
   class (MpiTestMethod), intent(inout) :: this
   real :: a(1,1), at(1,1)
   integer :: comm

   a = 1 ! preconditions

   comm = this%getMpiCommunicator()
   call transpose(comm, a, at)

   @mpiAssertEqual(1, at)
end subroutine testTranspose_trivial
```

./Exercises/SimpleMpiTest/TestTranspose.pf

Within an MPI test, failing assertions (either type) will

- Indicate number of processes used in test
- Rank(s) of process(es) that detected failure

Using assertions with MPI tests

Because the various assert annotations issue `RETURN` statements, care must be taken to avoid unintendend hangs on subsequent statements:

Using assertions with MPI tests

Because the various assert annotations issue RETURN statements, care must be taken to avoid unintendend hangs on subsequent statements: The following can hang under some circumstances:

```
@assertEqual(1., x)
@assertEqual(0., y)
```

- Some processes may not reach second assert statement.
- Use @mpiAssertEqual instead

Using assertions with MPI tests

Because the various assert annotations issue `RETURN` statements, care must be taken to avoid unintendend hangs on subsequent statements: The following can hang under some circumstances:

```
@assertEqual(1., x)
call MPI_Barrier(comm, ier)
```

- Some processes may not reach barrier statement.
- Use `@mpiAssertEqual` instead

Using assertions with MPI tests

Because the various assert annotations issue RETURN statements, care must be taken to avoid unintendend hangs on subsequent statements: The following can hang under some circumstances:

```
if (x > 0) then
   @mpiAssertEqual(3, i)
endif
```

- some processes may not reach implict barrier in assert.

Using assertions with MPI tests

Because the various assert annotations issue RETURN statements, care must be taken to avoid unintendend hangs on subsequent statements:

Best practice:

- Do *not* place assertions within conditional logic
- Use just 1 assertion per test procedure
- Use @mpiAssertTrue
- Use a local variable to hold expected value on each process, e.g.,

```
if (rank == 0) then
   expectedSum = 1.5
else
   expectedSum = 0.0
end if
@mpiAssertEqual(expectedSum, foundSum)
```

Exercise 2: Extend transpose tests

In this exercise, we will test an existing implementation of an MPI-based transpose procedure that accepts 3 arguments:

- `comm` - (input) MPI communicator[2]
- `a` - (input) array with columns distributed across processes
- `at` - (output) transpose of `a` with columns distributed

[2] **SUT must not use** `MPI_COMM_WORLD`

Exercise 2: Extend transpose tests

In this exercise, we will test an existing implementation of an MPI-based transpose procedure that accepts 3 arguments:

- `comm` - (input) MPI communicator[2]
- `a` - (input) array with columns distributed across processes
- `at` - (output) transpose of `a` with columns distributed

For example, if we have

$$A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \qquad A^T = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

[2] **SUT must not use `MPI_COMM_WORLD`**

Exercise 2: Extend transpose tests

In this exercise, we will test an existing implementation of an MPI-based transpose procedure that accepts 3 arguments:

- `comm` - (input) MPI communicator[2]
- a - (input) array with columns distributed across processes
- at - (output) transpose of a with columns distributed

For example, if we have

$$A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \qquad A^T = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Then for 2 processes, we have

- `a(:,1) = [1,2]` on p_0
- `a(:,1) = [3,4]` on p_1
- `at(:,1) = [1,3]` on p_0
- `at(:,1) = [3,4]` on p_1

[2] **SUT must not use** `MPI_COMM_WORLD`

Exercise 2a: Build transpose tests

1. Set `$PFUNIT` to the MPI build installation
2. Build and run tests
    1. Change directory to `./Exercises/SimpleMpiTest`
    2. `make tests`
    3. Verify that 2 tests ran successfully.
    4. Q: How many processes does the 1st test use?
    5. Q: How many processes does the 2nd test use?

Exercise 2b: Build broken test

1. Uncomment the line for `BrokenTest_mod_suite` in ‘testSuites.inc’
2. `make tests`

Exercise 2b: Build broken test

1. Uncomment the line for `BrokenTest_mod_suite` in ‘testSuites.inc’
2. `make tests`

   You should now see something like

```
...F
Time:         0.009 seconds

 Failure in: testBroken_2x2[npes=2]
   Location: [BrokenTest.pf:33]
expected +5.000000 but found: +1.000000;     difference: |+4.000000| > tolerance
     :+0.000000;  first difference at element [1, 1]. (PE=0)

 Failure in: testBroken_2x2[npes=2]
   Location: [BrokenTest.pf:33]
expected +5.000000 but found: +2.000000;     difference: |+3.000000| > tolerance
     :+0.000000;  first difference at element [1, 1]. (PE=1)

 FAILURES!!!
Tests run: 3, Failures: 1, Errors: 0
--------------------------------------------------------------------------
mpirun noticed that the job aborted, but has no info as to the process
that caused that situation.
--------------------------------------------------------------------------
*** Encountered 1 or more failures/errors during testing. ***
make: *** [tests] Error 128
```

Exercise 2b: Build broken test

1. Uncomment the line for `BrokenTest_mod_suite` in 'testSuites.inc'
2. `make tests`
3. Deactivate `BrokenTest_mod_suite`

A better test?

The existing tests are unsatisfactory in that they are hardwired to specific numbers of processes. Here we attempt to build a test that should work on arbitrary counts.

A better test?

The existing tests are unsatisfactory in that they are hardwired to specific numbers of processes. Here we attempt to build a test that should work on arbitrary counts.

We facilitate this by using a generator function that can be used to generate *synthetic* array elements that are *independent* of parallelism:

$$a_{ij} = a^T_{ji} = n_p * i + j$$

A better test?

The existing tests are unsatisfactory in that they are hardwired to specific numbers of processes. Here we attempt to build a test that should work on arbitrary counts.

We facilitate this by using a generator function that can be used to generate *synthetic* array elements that are *independent* of parallelism:

$$a_{ij} = a^T_{ji} = n_p * i + j$$

```
  real function arrayEntry(i,j,np) result(a)
     integer, intent(in) :: i
     integer, intent(in) :: j
     integer, intent(in) :: np

     a = np*j + i

  end function arrayEntry
```

./Examples/SimpleMpiTest/TestTranspose2.pf

Using the generator

```
        p = this%getProcessRank()
        do i = 1, npes
           j = p + 1
           a(i,1) = arrayEntry(i,j,npes)
        end do

        do i = 1, npes
           j = p + 1
           at_expected(i,1) = arrayEntry(j,i,npes)
        end do

        call transpose(comm, a, at_found)

        @mpiAssertEqual(at_expected, at_found)
```

./Examples/SimpleMpiTest/TestTranspose2.pf

Exercise 2c: Activate generalized test

1. Uncomment 3rd suite in ‘testSuites.inc’
2. `make tests`
   - You should now see 4 tests run
   - New test procedure was run twice: on 1 and 2 pes respectively.
3. Edit ‘TestTranspose2.pf’ to have the test run on 1,2,3, and 4 processes.
4. `make tests`
   - You should now see 6 tests run
5. Q: What happens when you include a case with 5 processes?

Fixtures

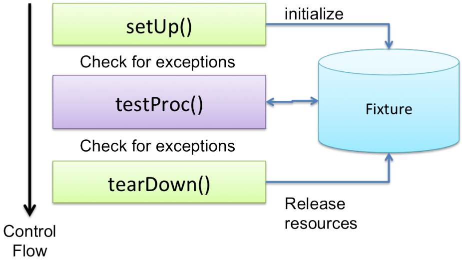
www.remodelista.com

Test Fixtures

A test *fixture* is any mechanism that allows a consistent initialization for test preconditions.

- Group of tests have same initial conditions
- Complex sequence of steps to create preconditions
- Ensures release of system resources (memory, files, ...)

Testing frameworks generally provide mechanisms to encapsulate the logic for test preconditions and cleanup. Usually this is in the form of procedures named setUp() and tearDown().



Simple fixture (unencapsulated)

Can be expedient to use a global variable or a temporary file as a quick-and-dirty fixture:

```
module SimpleFixture_mod
   use pFunit
   use Reader_mod

contains

   @before
   subroutine init()
      open(10,file='tmp.dat',status='new')
      write(10) 1
      write(10) 2
      close(10)
   end subroutine init

   @after
   subroutine done()
      open(10,file='tmp.dat',status='unknown')
      close(10, status='delete')
   end subroutine done

   ...
```

Simple fixture (contd́)

- `@before` indicates procedure to run before each test in file
  - Convention is to call procedure setUp()
- `@after` indicates procedure to run after each test in file
  - Convention is to call procedure tearDown()
- Because no arguments are passed to procedures, fixture data must be in the form of global variables (module, common) or the file system (file)
  - Dangerously close to violating rule: "No Side Effects!"

barksandblooms1.blogspot.com

Trapping illegal input

Consider the following code snippet:

```
subroutine checkInputs(n)
   if (n <= 0) then
      call stopModel('n must be positive')
   endif
end subroutine checkInput
```

Trapping (cont'd)

We want to *test* that the application traps bad values

Trapping (cont'd)

We want to *test* that the application traps bad values
So we start writing a test ...

```
@test
subroutine testNegativeN()
   call checkInputs(-1)
   ? what do we check ?
end subroutine
```

Trapping (cont'd)

We want to *test* that the application traps bad values
So we start writing a test ...

```
@test
subroutine testNegativeN()
   call checkInputs(-1)
   ? what do we check ?
end subroutine
```

We immediately encounter some difficulties

- The test terminates execution
- We don't have anything to check inside the test anyway

Trapping (cont'd)

What if istead the application looked like this:

```
subroutine checkInputs(n)
   if (n <= 0) then
      ! send signal ("exception") to pFUnit
      call throw('n must be positive')
      return
   endif
end subroutine checkInput
```

And the test:

```
@test
subroutine testNegativeN()
   call checkInputs(-1)
   @assertExceptionRaised('n must be positive')
end subroutine
```

Trapping (cont'd)

Of course we now have new problems:

- We have introduced a dependency on pFUnit in the application
- When not testing the code does not stop.

My usual kludge to deal with this is:

```
#ifdef USE_PFUNIT
subroutine stopModel(message)
   use pFUnit_mod, only: throw
   character(len=*), intent(in) :: message
   call throw(message)
end subroutine stopModel
#else
subroutine stopModel(message)
   use pFUnit_mod, only: throw
   character(len=*), intent(in) :: message
   print*,message
   stop
end subroutine stopModel
#endif
```

Trapping (cont'd)

Summary

- Use `throw` to signal undesired state
- Use `@assertExceptionRaised` to detect signal
- Use preprocessor to limit dependency on framework
  - **Must not stop execution in testing configuration.**
  - Should not depend on pFUnit in production configuration

Outline

Exception class

Role: Used to notify/detect "undesired" states during execution. *Limited* emulation of exceptions provided by other high-level languages (C++, Java, Python, etc).

Implementation:

- Manages a global, private stack of Exception objects.
- Each Exception object has a message, and a location (file+line).

Exception API

```
subroutine throw(message[, location])

logical function catch([preserve])

logical function catch(message, [preserve])

type (Exception) function catchNext([preserve])
```

Assert modules

Role: Used to express *intended/expected* relationships among variables.

Implementation:

- Heavily overloaded suite of procedures with consistent style for interface.
- When the intended relationship does not hold, the layer pushes a self-explanatory Exception onto the global exception stack.

Logical Assertions

```
call assertTrue(condition)
call assertFalse(condition)
call assertAny(conditions)
call assertAll(conditions)
call assertNone(conditions)
call assertNotAll(conditions)
```

String Assertions

```
call assertEqual(expected, found)
```

Integer Assertions

```
call assertEqual(expected, found)
```

- Overloaded for up to rank 2 (Need more? send a support request!)
- Only supports default KIND

The following are only supported for scalars:

```
call assertLessThan(a, b) ! a < b
call assertLessThanOrEqual(a, b) ! a <= b
call assertGreaterThan(a, b) ! a > b
call assertGreaterThanOrEqual(a, b) ! a >= b
```

API - AssertEqual (Real)

Compare two values and throw exception if different

$$|a - b| > \delta$$

```
call assertEqual(expected, found[, tolerance])
```

- Uses *absolute* error (as opposed to *relative* error)
- Overloaded for multiple KINDs (32 and 64 bit)
- Overlaoded for multiple ranks (up through 5D)
- Optional tolerance – default is *exact* equality
- Uses L_∞ norm
- To reduce exponential number of overloads:
  - KIND(expected) <= KIND(found)
  - KIND(tolerance) == KIND(found)
  - RANK(expected) == RANK(found) or scalar

Example message:

```
expected: +1.000000 but found: +3.000000;
    difference: |+2.000000| > tolerance:+0.000000.
```

API - Assert variants (Real)

```
call assertLessThan(expected, found)
call assertGreaterThan(expected, found)
call assertLessThanOrEqual(expected, found)
call assertGreaterThanOrEqual(expected, found)
```

If relative tolerance is desired:

$$\text{If } \frac{|a-b|}{|a|} > \delta \text{ then fail}$$

```
call assertRelativelyEqual(expected, found[, tolerance])
```

API - AssertEqual (Complex)

Compare two values and throw exception if different

$$|a - b| > \delta$$

```
call assertEqual(expected, found[, tolerance])
```

- Overloaded for multiple KINDs (32 and 64 bit)
- Overlaoded for multiple ranks (up through 5D)
- Optional tolerance – default is *exact* equality
- To reduce exponential number of overloads:
  - KIND(expected) $<=$ KIND(found)
  - KIND(tolerance) $==$ KIND(found)
  - RANK(expected) == RANK(found) or scalar

Miscellaneous other Assert procedures

```fortran
call assertIsNaN(x)      ! single/double
call assertIsFinite(x) ! single/double
call assertExceptionRaised()
call assertExceptionRaised(message)
call assertSameShape(expectedShape, foundShape)
```

Outline

1. Introduction
2. Introduction to pFUnit
   - pFUnit overview
   - Build and Install
   - Simple examples
   - API: Exceptions and Assertions
   - Parser: Basic
3. References and Acknowledgements

Annotations: Miscellaenous

- `@suite`

  Overrides default name for generated function which constructs test suite for the input file. Default is `<base>_suite` for file with external test procedures, and `<module_name>_suite` for files that contain a module.

- `@before`

  Indicates next line begins a setUp() procedure for subsequent test procedures.

- `@after`

  Indicates next line begins a tearDown() procedure for subsequent test procedures.

Annotations: @assert*

```
@assert*(...)
```

1. Calls corresponding Fortran assert procedure
2. Inserts argument for file & line number
3. Inserts conditional return if exception is thrown

For example, if line 100 of file 'myTests.pf' is:

```
@assertEqual(x, y, tolerance)
```

Expands to

```
!@assertEqual(x, y, tolerance)
call assertEqual(x, y, tolerance, &
  & SourceLocation('myTests.pf', 100))
if (anyExceptions()) return
```

Annotations @test

```
@test
@test(<options>)
```

- Indicates that next line begins a new test procedure
- Appends test procedure in the file's TestSuite
- Accepts the following options:
  - `ifdef=<token>` Enables conditional compilation of test
  - `npes=[<list-of-integers>]` Specifies that test is to run in a parallel context on the given numbers of processes.
  - esParameters={expr} Run this test once for each value in expr. Expr can be an explicit array of TestParameter's or a function that returns sech an array.
  - `cases=[<list-of-integers>]` Alternative mechanism for specifying test parameters where a single integer is passed to the test constructor.

Outline

References

- pFUnit: http://sourceforge.net/projects/pfunit/
- Tutorial materials
  - https://modelingguru.nasa.gov/docs/DOC-1982
  - https://modelingguru.nasa.gov/docs/DOC-1983
  - https://modelingguru.nasa.gov/docs/DOC-1984
- TDD Blog https://modelingguru.nasa.gov/blogs/modelingwithtdd
- *Test-Driven Development: By Example* - Kent Beck
- Mller and Padberg,"About the Return on Investment of Test-Driven Development," http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf
- *Refactoring: Improving the Design of Existing Code* - Martin Fowler
- JUnit http://junit.sourceforge.net/

Acknowledgements

- This work has been supported by NASA's High End Computing (HEC) program and Modeling, Analysis, and Prediction Program.
- Many thanks to team members Carlos Cruz and Mike Rilee for helping with implementation, regression testing and documentation.
- Special thanks to members of the user community that have made contributions.
  - Sean Patrick Santos
  - Matthew Hambley
  - Evan Lezar